Proposta de uma Arquitetura *Multi-Threading* Voltada para Sistemas Multi-Processos.

Ronaldo A. L. Gonçalves[1]
Departamento de Informática / UEM / Brazil
e-mail: ronaldo@din.uem.br

Philippe O. A. Navaux
Instituto de Informática / UFRGS / Brazil
e-mail: navaux@inf.ufrgs.br

Resumo

Este artigo propõe o desenvolvimento de uma arquitetura *multi-threading* capaz de extrair tanto o paralelismo ao nível de instruções quanto aquele disponível entre os diferentes processos executados pelos sistemas operacionais nas estações de trabalho compartilhadas e servidores de rede. A arquitetura proposta alivia o sistema operacional das atividades mais onerosas em consumo de tempo de cpu, tais como escalonamento e troca de contexto entre processos. Aqui são apresentados os principais componentes da arquitetura, bem como os algoritmos básicos a serem executados pelos estágios do *pipeline* superescalar.

palavras-chaves: superescalar, *multi-threading*, gerenciamento de processos

Abstract

This paper proposes the development of a multi-threading architecture in order to be able to extract the instruction level parallelism as well as that parallelism avaiable between different processes executed by operating systems on both shared workstations and network servers. The architecture proposed alleviates the operating systems of the more onerous activities in waste of cpu time, such as scheduling and contextual change between processes. The main components of architecture are presented here, as well as the basic algorithms used by stages of the superscalar pipeline.

key-words: superscalar, *multi-threading*, processes management

[1] Doutorando no CPGCC (Instituto de Informática) da Universidade Federal do Rio Grande do Sul
e-mail: ronaldog@inf.ufrgs.br

Proposta de uma Arquitetura *Multi-Threading*

Voltada para Sistemas Multi-Processos.

1. Introdução.

Atualmente, a complexidade das aplicações tem exigido processadores mais rápidos. Neste contexto, os processadores superescalares têm sido a tecnologia adotada comercialmente, pois são capazes de extrair paralelismo a partir de programas seqüenciais, aumentando o desempenho das aplicações comumente disponíveis. Apesar desta vantagem, estes processadores falham em não explorar o paralelismo existente entre aplicações distintas, que pode ser muito superior ao paralelismo ao nível de instrução. Entre os processadores superescalares atuais mais comumente usados podem ser destacados: *Pentium* [ANDE95], *PowerPC* [CHAK94, DIEP95] e MIPS R10000 [MIPS95].

Desta forma, muitas pesquisas vem sendo realizadas para desenvolverem arquiteturas *multi-threading*, que são capazes de executar paralelamente várias instruções provenientes de diferentes fluxos de controles. Apesar desta tecnologia ser promissora, o desenvolvimento de arquiteturas *multi-threading* tem sido desmotivado pela falta de aplicações com múltiplas *threads*, pois a maioria esmagadora das aplicações existentes é seqüencial.

Mas por outro lado, grande parte dos processadores atuais é utilizada em estações de trabalho ou servidores de rede, com seus recursos compartilhados por diferentes aplicações. Estas aplicações em conjunto com o próprio sistema operacional formam um conjunto de processos em execução, que pode ser chamado de sistema multi-processos, onde muito do tempo de execução destes processadores é gasto com o gerenciamento destes processos, principalmente com o escalonamento e as trocas de contexto.

Visando desenvolver processadores voltados para a execução de sistemas multi-processos, o presente trabalho propõe uma arquitetura que não deixa de ser *multi-threading* e nem tão pouco superescalar, mas caracteriza-se pela habilidade adicional de extrair o paralelismo existente entre processos, além de executar diretamente pelo *hardware*, muitas das operações (onerosas em consumo de tempo de cpu) normalmente executadas pelo sistema operacional.

Este artigo está organizado da seguinte forma: o capítulo 2 introduz uma visão geral sobre o estado da arte em arquiteturas *multi-threading*; o capítulo 3 apresenta e descreve o funcionamento da arquitetura proposta, e os capítulos 4 e 5 apresentam as conclusões e as referências bibliográficas, respectivamente.

2. Arquiteturas *Multi-Threading*: O Estado da Arte.

As arquiteturas *multi-threading* podem ser vistas como uma evolução das arquiteturas superescalares tradicionais, com o aproveitamento de tecnologias derivadas dos multiprocessadores. Na prática, existem várias formas de se estruturar uma arquitetura *multi-threading* [TULL95].

Laudon, Gupta e Horowitz [LAUD94] propuseram uma técnica de execução *multi-threading*, chamada *Interleaving*, que pode ser aplicada em processadores superescalares tradicionais, para permitir que os mesmos executem aplicações tanto *single-thread* quanto *multi-threading*, sem a adição de *hardware*. No extremo oposto, Govindarajan e Nemawarkar [GOVI92] projetaram um multiprocessador chamado SMALL, composto de várias unidades de processamento independentes, para executar *multi-threading*.

Mas a maioria das propostas de arquiteturas *multi-threading* se baseia na replicação das estruturas de armazenamento existentes em uma arquitetura superescalar tradicional, para poder suportar múltiplas *threads*. Desta forma, Hirata e outros [HIRA92] desenvolveram uma arquitetura *multi-threading* que utiliza bancos de registradores, filas de instruções e contadores de programa, todos específicos para cada *thread* em execução. Também, Wallace, Calder e Tullsen [WALL98] projetaram uma arquitetura *multi-threading* com várias tabelas de renomeação de registradores, para executar ambos os caminhos dos desvios condicionais, quando houver mais *hardware* do que *threads* disponíveis.

De uma forma geral, uma arquitetura *multi-threading* básica pode ser vista na figura 1, onde nota-se que a diferença principal desta arquitetura com uma arquitetura de multiprocessador está no compartilhamento das unidades funcionais (UF1...UFm) e memórias (D & I *Caches*). Já as estruturas de armazenamento da arquitetura (filas de instruções e arquivo de registradores) são separadas fisicamente, ou pelo menos logicamente, formando *slots*, para abrigar diferentes *threads*.

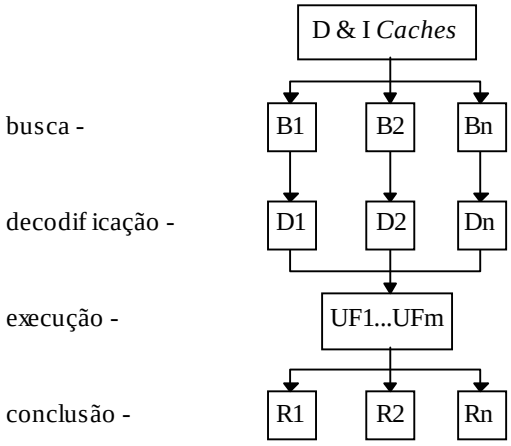


Figura 1: Arquitetura *multi-threading* básica

Os dados manipulados pelas *threads* formam contextos distintos dentro da arquitetura, onde o arquivo de registradores deve ser grande o suficiente para armazenar muitos contextos. O objetivo principal de uma arquitetura *multi-threading* é aumentar substancialmente a utilização do processador na presença de longas latências, como nas ocorrências de *cache miss* e de pouco paralelismo disponível por *thread*, devido as dependências de dados [LAUD94]. Quanto maior é o número de *threads* disponíveis, mais fácil é para mascarar estas latências, pois o número de instruções úteis disponíveis também é maior [PARK91].

3. Arquitetura Proposta

A arquitetura aqui proposta é vista na figura 2, que mostra seus principais componentes funcionais e estruturais. A mesma possui um *pipeline* superescalar com 5 estágios funcionais (busca, decodificação, execução, término e conclusão), providos de técnicas usuais de previsão de desvios, execução especulativa, renomeação simplificada de registradores e execução fora-de-ordem. Os principais componentes da arquitetura são:

- Uma fila de *frames* disponíveis (FD), para conter os identificadores dos *frames* disponíveis para serem alocados para os processos.

- Uma fila de processos (FP), para conter descritores de todos os processos prontos, suspensos ou mortos (mas que ainda não foram removidos do *hardware*). Cada descritor compõe-se de: contador de programa (CP), *time-slice* (TS) fornecido pelo sistema operacional, identificador do *frame* (Fid), e o *status* do processo (Stt), que indica o seu estado atual (pronto, suspenso ou morto). Neste artigo, "descritor de processo" é tratado apenas como "processo".

- Um conjunto de *slots*, cada um contendo uma fila de instruções (FI), uma fila de processos ativos (FA), e um um registrador de descritor de processo (RDP), que contém uma cópia do descritor do processo que está sendo atualmente buscado.

- Um conjunto de filas de remessa (FRm), uma para cada unidade funcional, que contém instruções prontas para serem remetidas para as unidades funcionais.

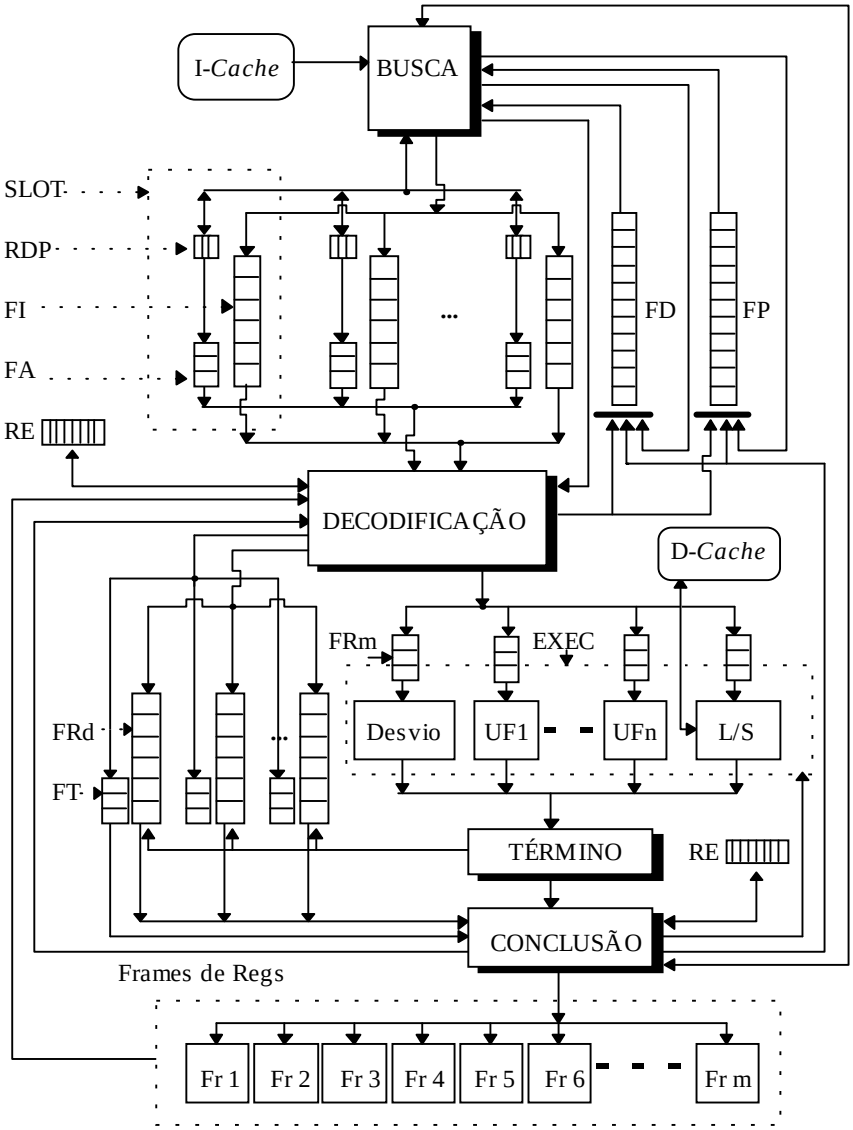


Figura 2: Arquitetura *multi-threading* proposta.

- Um conjunto de unidades funcionais compartilhadas por todos os *slots*.

- Dois registradores de escalonamento (RE), que auxiliam no escalonamento de instruções prontas, durante os estágios de decodificação e conclusão.

- Um conjunto de registradores, divididos em *frames* (Fri) separados, para armazenar os contextos dos processos. Cada *frame* é identificado por um Fid diferente.

- Um conjunto de filas de reordenação (FRd), cada uma associada a um *slot* diferente, que mantém a ordem original das instruções despachadas.

- Um conjunto de filas de processos em trânsito (FT), que contém os processos que possuem instruções nas FRds. Cada FT é associada à uma FRd, e conseqüentemente, à um *slot* diferente.

Todas as filas utilizadas na arquitetura (FP, FD, FA, FI, FRd, FT e FRm) são manipuladas no modelo FIFO (*first in first out*), muito embora as entradas das filas FP e FRd possam ser atualizadas (mas não inseridas ou removidas) aleatoriamente. Os estágios da arquitetura proposta são discutidos nas próximas seções.

3.1. Estágio de Busca.

As principais funções do estágio de busca são: busca de instruções da *cache*, previsão de desvios, troca de contexto e detecção e execução de instruções privilegiadas. A figura 3 mostra o estágio de busca e alguns dos seus relacionamentos.

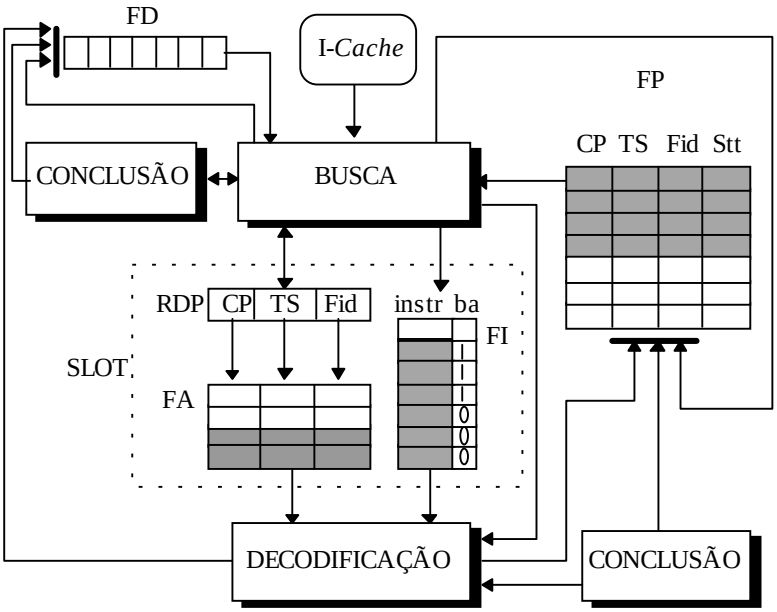


Figura 3: Esquema do estágio de busca

Durante a busca de instruções na *cache*, este estágio tenta prover uma quantidade suficiente de instruções para o restante do *pipeline*, buscando um bloco de instruções por vez, para cada *slot* da arquitetura. Este bloco deve conter pelo menos N*T instruções, onde N indica o número de *slots* existentes e T o tempo médio necessário para buscar o bloco na *cache*, assim como sugerido por Hirata e outros em [HIRA92]. Isto tende a favorecer que cada *slot* permaneça com instruções até a próxima busca. Obviamente, o cálculo de T deve considerar que muitos desvios condicionais, encontrados dentro do referido bloco, são previstos como tomados, ocasionando vários redirecionamentos da busca.

A escolha do *slot* a ser preenchido é feita no estilo *round-robin*, mas futuramente outras políticas serão avaliadas, pois segundo Tullsen [TULL96], a escolha de uma política de busca correta é um dos fatores de maior importância para a obtenção de alto desempenho em arquiteturas *multi-threading*. Uma vez selecionado o *slot*, a busca é feita a partir do endereço contido no campo CP do registrador RDP, e as instruções buscadas são inseridas na FI.

Para suportar execução especulativa, o estágio de busca prevê desvios condicionais utilizando um mecanismo baseado em um *bit* de história. Se o desvio nunca foi executado anteriormente, a previsão é sempre como desvio tomado. Este mecanismo obtém 90,23% de acerto nas previsões, segundo Smith [SMIT81]. Mas a principal vantagem deste mecanismo, para arquiteturas *multi-threading*, é que o *bit* de história é associado diretamente nas instruções de desvios, dentro da *cache*, eliminando assim a utilização de tabelas de previsão específicas para cada *thread*, o que causa um aumento considerável nas lógica e estruturas de armazenamento. O funcionamento básico do estágio de busca é mostrado no algoritmo a seguir.

```
Algoritmo_Busca;
Para cada_ciclo Faça
1.  Seleciona_slot (ns) ;    /* round robin */
2.  Se vazio(ns) ou (RDP.TS ≤ 0)
        3.  Então troca_contexto (ns) ;
    Fim-Se ;
4.  Busca_instruções_para_o_processo (RDP.CP(ns), flag) ;
5.  Se ocorrer_cache_miss (flag)
        6.  Então  troca_contexto (ns) ;
                7.  volta_passo_4 ;
    Fim-Se ;
8.  Para cada_instrução_buscada Faça
        9.  Caso instrução Seja:
            10. branch: prevê_desvio ;
            11. create: cria_processo ;
            12. kill: mata_processo ;
            13. suspend: suspende_processo ;
            14. resume: reassume_processo ;
            Fim-Caso ;
        15. Coloca_instrução_na_FI (ns);
    Fim-Para ;
Fim-Para ;          /* repeat forever */
```

A troca de contexto, em um determinado *slot*, ocorre quando ele ou está vazio (no início da computação) ou quando a busca das instruções do processo atual é interrompida. Esta operação não requer o esvaziamento do *slot* para o posterior preenchimento com outro contexto, como normalmente tem sido utilizado nas arquiteturas *multi-threading* propostas. Nesta arquitetura, as instruções do novo contexto são concatenadas, na seqüência, com as instruções do contexto antigo, ainda remanescentes no *slot*. O efeito deste procedimento é a antecipação da busca do próximo contexto, e por conseqüência, a maximização da utilização das filas de instruções.

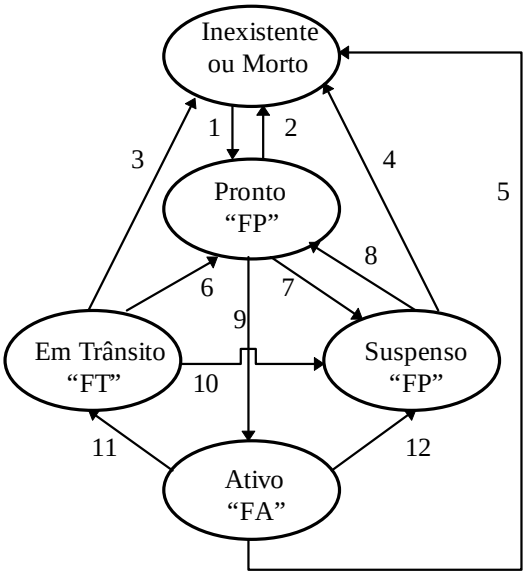
Sempre que é necessária a troca de contexto, o estágio de busca escalona um processo da FP (retira o primeiro da fila) e verifica o seu campo de *status* (Stt). Se o *status* indicar que o processo está “morto”, ele é descartado e seu *frame* é inserido na FD. Se o *status* indicar que o processo está “suspenso”, ele é reinserido na FP. Mas se o *status* indicar que o processo está “pronto”, ele é inserido na FA (com exceção do campo *status*) e uma cópia do mesmo descritor é mantido no RDP, para as futuras verificações do estágio de busca.

O controle de quais contextos estão inseridos no *slot* é feito através da FA. A FA contém uma entrada para cada contexto ativo dentro do *slot*, que é inserida pela unidade de busca, quando houver troca de contexto no respectivo *slot*. As novas instruções são inseridas na FI com o *bit* alternador inverso ao *bit* alternador do contexto anterior (veja figura 3). De uma forma geral, a troca de contexto ocorre nos seguintes casos:

- durante a ocorrência de i-*cache miss*.
- durante a ocorrência de instruções privilegiadas.
- final de *time-slice* do processo.

Existem quatro instruções privilegiadas que podem ser solicitadas pelo sistema operacional e executadas diretamente pelo estágio de busca. O sistema operacional gerencia os processos e o *hardware* apenas provê facilidades para tal. As instruções privilegiadas e suas semânticas são resumidas a seguir, e a figura 4 mostra os possíveis estados e transições de um processo ao nível de arquitetura.

- ***create*** **op1 op2 rde :** cria um novo processo - Retira o Fid da FD e insere um novo processo na FP, contendo: o Fid, o CP (deve estar em op1) e o TS (deve estar em op2). Note que através de diferentes *time-slices*, os processos criados podem ter diferentes prioridades. O Fid é retornado em rde (como identificador do processo).
- ***kill*** **op1 rde :** mata um processo - O processo (cujo Fid deve estar em op1) é eliminado da arquitetura. O rde retorna o sucesso da operação.
- ***suspend*** **op1 op2 rde :** suspende um processo - O processo (cujo identificador deve estar em op1) tem seu *status* marcado como "suspenso" e quando ele for escalonado é reinserido na FP. O rde retorna o sucesso da operação.
- ***resume*** **op1 rde :** reassume um processo suspenso - O processo (cujo identificador deve estar em op1) que está marcado como "suspenso" é simplesmente remarcado como "pronto". O rde retorna o sucesso da operação.



Transições

1 - *create*
2 - *kill*
3 - *kill* ou
execução irregular
4 - *kill*
5 - *kill*
6 - final de execução
parcial do contexto
7 - *suspend*
8 - *resume*
9 - escalona (troca
de contexto)
10 - *suspend*
11 - decodifica (inverte
bit alternador)
12 - *suspend*

Figura 4: Diagrama de estados dos processos na arquitetura proposta

3.2. Estágio de Decodificação.

De uma forma geral, o estágio de decodificação é responsável por três principais atividades, que são executadas em conjunto: escalonamento das instruções dos *slots*, decodificação (incluindo leitura dos operandos) e despacho para as filas de remessa das unidades funcionais. As maiores estruturas acessadas por este estágio são: FIs, FAs, FRds, FRms, FT e RE. A figura 5 mostra um esquema simplificado manipulando uma única estrutura de cada. Este estágio decodifica as instruções contidas nas primeiras entradas das FIs dos *slots*, e despacha somente as instruções que estão prontas. Isto é feito para evitar a saturação das filas de remessa com instruções não remessíveis, já que em arquiteturas *multi-threading* a quantidade de instruções prontas é bem maior do que em arquiteturas superescalares tradicionais.

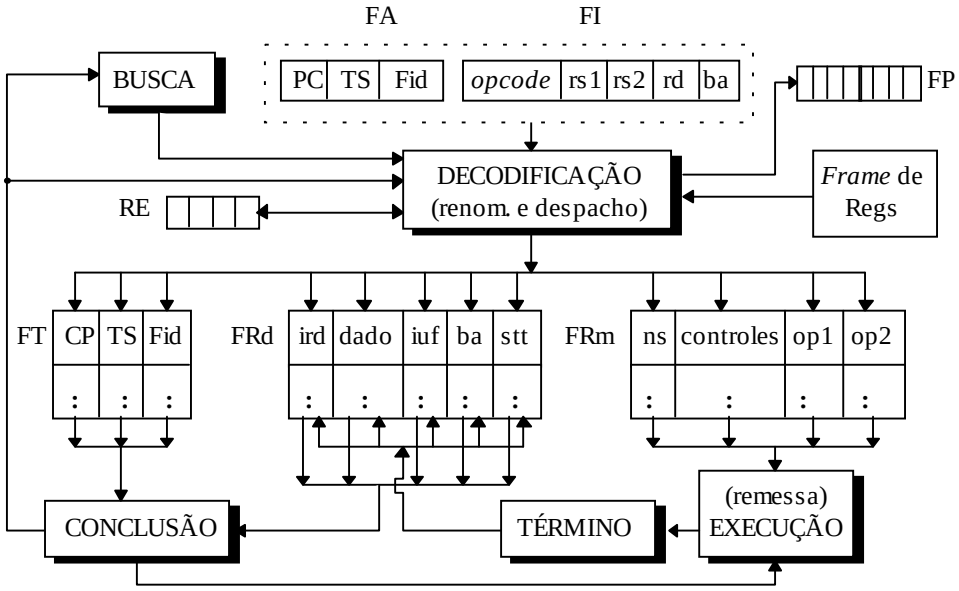


Figura 5: Esquema do estágio de decodificação

Quando uma instrução é despachada, a FI associada é deslocada e a próxima instrução é visualizada no início da FI. Quando o estágio de decodificação detecta uma inversão do *bit* alternador (ba), a FA também é deslocada, e a primeira entrada é então inserida no final da FT associada ao *slot* em questão. O funcionamento básico do estágio de decodificação é mostrado no algoritmo a seguir.

```
Algoritmo Decodificação
Para cada_ciclo Faça
   Repita
   1. Seleciona_slot (ns) ;          /* round robin ponderado */
   2. Identifica_instrução_e_tipo (ns, Fid, instr, tipo) ;
   3. Se   instrução_da_FI_está_pronta (Fid, instr) e
      4. existe_entrada_na_FRm (tipo, entrada1) e
      5. existe_entrada_na_ FRd (ns, entrada2)
   6. Então  lê_operandos (Fid, op1, op2) ;
         7. despacha_instrução_na_FRm (entrada1, ns, instr, info) ;
         8. insere_entrada_na_FRd (entrada2, iuf, info) ;
      Fim-Se ;
   Até preencher_largura_de_decodificação_despacho ;
Fim-Para ;          /* repeat forever */
```

O algoritmo de escalonamento, chamado aqui de "*round-robin* ponderado", utiliza um registrador RE contendo um identificador para cada *slot* da arquitetura. Para cada identificador de *slot*, existe também um *bit* de despacho, que informa se alguma instrução daquele *slot* já foi despachada alguma vez. Inicialmente, todos os *bits* contém 0 (zero), indicando que não houve nenhum despacho de nenhum *slot*. O algoritmo percorre o RE circularmente, iniciando pela primeira entrada do registrador, e para cada identificador de *slot* ele verifica se existe instrução pronta no referido *slot*. Para saber quais instruções estão prontas, este estágio verifica os *busy-bits* (*bits* de ocupação) dos registradores fontes (rs1 e rs2) de cada instrução, dentro dos respectivos *frames* de registradores (indicado pelo campo Fid lido da FA).

Para cada instrução pronta que é localizada, é verificado se existe entrada livre na fila de remessa para onde a instrução deve ser despachada. Se sim, a instrução é selecionada para o despacho. O algoritmo continua até preencher a largura de despacho ou até não haver mais instruções prontas. Então, as instruções selecionadas têm seus operandos lidos e são então despachadas, juntamente com o número do *slot* e os controles decodificados. Para cada *slot* em que uma ou mais instruções são despachadas, o *bit* de despacho é marcado com 1 (um).

Após o despacho, o RE é reorganizado de forma que os *slots* que já tiveram instruções despachadas são colocados no final do RE, para permitir que os outros *slots* sejam os próximos a serem escalonados. Se existe dificuldade em se encontrar instruções prontas nos diferentes *slots*, mais de uma instrução pode ser despachada de um mesmo *slot* em um mesmo ciclo (o mesmo *slot* é varrido novamente no RE), desde que estejam prontas. Este algoritmo favorece os processos com maior número de instruções prontas, ao mesmo tempo que cede a todos os *slots* a oportunidade de despachar. Quando todos os *bits* de despacho do registrador RE estão com 1 (um), eles são todos reinicializados novamente com 0 (zero).

Para cada instrução despachada, o registrador destino (rd) tem o seu *busy-bit* marcado como "ocupado" (para indicar que o dado ainda não está pronto). Além disso, é inserido uma entrada na FRd (associada ao *slot* da onde a instrução foi despachada) contendo o identificador do registrador destino (ird), um campo para o resultado da instrução (dado), o identificador da unidade funcional que vai fornecer o resultado (iuf), o *bit* alternador (ba) e um campo de *status* (stt) marcado com instrução "não terminada".

As FRds servem para conter os dados gerados pelas instruções, o que garante o controle das falsas dependências, pois isto elimina os conflitos entre os registradores destinos. Posteriormente, no estágio de conclusão, o dado da FRd é retirado e gravado no registrador destino que tem o seu *busy-bit* marcado como "pronto".

3.3 Estágio de Execução.

Neste estágio, cada unidade funcional executa as instruções de sua respectiva FRm, retirando-as em ordem, através de deslocamento. Não existe nenhum outro tipo de escalonamento dinâmico, aqui neste estágio, pois as instruções já foram escalonadas durante o despacho. Se uma instrução causa uma exceção, tal como divisão por zero ou acesso não permitido, o resultado da instrução é enviado para o estágio de término como um código de erro. O funcionamento básico de cada unidade funcional é mostrado no algoritmo a seguir.

Algoritmo Execução ;
Repita
1. Retira_instrução_do_*buffer*_de_remessa (ns) ;
2. Executa_instrução (instr, resultado) ;
3. Envia_resultado_para_término (ns, resultado) ;

Fim-Repita ; /* repeat forever */

3.4. Estágio de Término.

Este estágio atualiza as entradas das FRds para cada instrução que termina a sua execução, da seguinte maneira: para cada unidade funcional que conclui a execução de uma instrução, a unidade de término procura (na ordem inversa a da inserção) na FRd associada ao *slot* da instrução concluída, a primeira entrada cujo campo "iuf" (identificador da unidade funcional) corresponde ao da unidade funcional em questão. Nesta entrada, ela grava o resultado da operação no campo "dado" e altera o campo de *status* "stt" para "instrução terminada".

Deve-se observar que as instruções despachadas para a mesma FRm são inseridas e retiradas em ordem. Também, as instruções dentro de uma FRd estão na mesma ordem em que estavam dentro do *slot*. Assim, para uma instrução, pertencente à um contexto posterior, terminar a sua execução antes de outra instrução, pertencente à um contexto anterior, elas devem ser executadas por unidades funcionais diferentes. Como as instruções dentro de uma FRd são pesquisadas pelo identificador da unidade funcional executora "iuf", jamais haverá coincidência entre dois registradores, com os mesmos identificadores, utilizados por duas instruções de contextos distintos. Isto garante a eliminação das falsas dependências.

Em caso de exceção, o campo "dado" é preenchido com código de erro, a ser tratado na conclusão. Também, em caso de instrução de desvio, o resultado obtido (tomado ou não tomado) é comparado com a previsão inicial. Se os valores diferem, o campo "stt" da FRd é marcado com "previsão incorreta" para ser tratado pelo estagio de conclusão. Caso contrário, o campo "stt" é marcado de forma normal, com "instrução terminada". O funcionamento básico do estágio de término é mostrado no algoritmo a seguir.

```
Algoritmo Término ;
Para cada_ciclo Faça
    Repita
    1. Seleciona_unidade_funcional_com_resultado_pronto (iuf);
    2. Recebe_resultado_da_unidade_funcional (iuf, ns, resultado) ;
    3. Insere_na_FRd_correta (ns, iuf, resultado) ;
    Até completar_largura_de_término ;
Fim-Para ;          /* repeat forever */
```

3.5. Estágio de Conclusão.

Para cada FRd, este estágio verifica se as primeiras instruções estão "terminadas", analisando o campo "stt". Aqui também é utilizado o escalonamento "*round robin* ponderado" apresentado na seção 3.2., para saber de quais FRds as instruções prontas devem ser retiradas primeiro, utilizando para isto um outro RE. Para cada instrução escolhida, a mesma é retirada da FRd, através de deslocamento, e o resultado (campo "dado") da instrução é gravado no registrador destino ("ird") do *frame* correspondente (campo "Fid" da primeira entrada da FT). Este mesmo registrador tem seu *busy-bit* marcado como "livre" e pode ser lido por outra instrução.

Quando termina a execução de um contexto (através da verificação da inversão do *bit* alternador na FRd correspondente), o processo é retirado da FT (deslocamento). Se por acaso, a unidade de busca solicitou que o processo fosse suspenso, seu campo de *status* é marcado para tal. Então, o processo é reinserido na FP. Neste estágio também são controladas a execução especulativa e ocorrência de exceções.

```
Algoritmo Conclusão ;
Para cada_ciclo Faça
   Repita
     1. Seleciona_FRd ;          /* round robin ponderado */
     2. Caso resultado Seja:
          3. exceção: descarta_processo_e_libera_frame_Fid ;
          4. previsão_incorreta: promove_redirecionamento_da_busca ;
        Fim-Caso ;
     5. grava_no_frame_correto (ns, resultado) ;
   Até completar_largura_de_conclusão ;
Fim-Para ;          /* repeat forever */
```

4. Conclusões e Trabalhos Futuros.

O presente trabalho propõe uma arquitetura *multi-threading*, voltada para a execução de múltiplos processos, existentes em grande quantidade nas estações de trabalho compartilhadas e servidores de rede. Esta arquitetura é capaz de explorar o paralelismo dentro de cada processo, executando instruções prontas fora-de-ordem, tão bem quanto explorar o paralelismo entre vários processos, escondendo latências e aumentando o desempenho final do sistema. As principais contribuições desta arquitetura são:

- A arquitetura é baseada em filas do tipo FIFO e algoritmos de escalonamento baseados em *round-robin*, provendo rapidez na execução e simplicidade da lógica de manipulação.

- A arquitetura contém um conjunto de instruções específicas para facilitar o desenvolvimento do sistema operacional. Estas instruções são simples e de fácil decodificação, e são resolvidas diretamente pelo estágio de busca de instruções. Isto permite ganho de tempo substancial, que tradicionalmente é perdido com gerenciamento de processos e troca de contexto ao nível de sistema operacional.

- A arquitetura suporta troca antecipada de contexto, tão bem quanto múltiplos processos compartilhando o mesmo *slot*, que são gerenciados através das filas de processos ativos, maximizando a utilização do *hardware*, que tradicionalmente abriga uma única *thread* por *slot*.

Como trabalhos futuros, os próximos passos serão a implementação de um simulador para a arquitetura proposta, a análise de desempenho e o balanceamento da arquitetura.

5. Referências Bibliográficas.

[ANDE95] Anderson, D. & Shanley, T., Pentium Processor System Architecture, MindShare, Inc., Addison-Wesley, Massachusetts, 433p., February, 1995.

[CHAK94] Chakravarty, D. & Cannon, C., PowerPC: Concepts, Architecture, and Design, J. Ranade Workstations Series, McGraw-Hill, USA, Inc.,p.363, 1994.

[DIEP95] Diep, T.A & Nelson, C. & Shen, J.P., Performance Evaluation of the PowerPC 620 Microarchitecture, Proceedings of the 22nd ISCA, Santa Margherita Ligure, Italy, June, 1995.

[GOVI92] Govindarajan, R. & Nemawarkar, S. S., SMALL: A Scalable Multithreaded Architecture to Exploit Large Locality, Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, Dec, 1992.

[HIRA92] Hirata, H. et all, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, Proceedings of the 19th Annual International Symposium on Computer Architecture, May 1992.

[LAUD94] Laudon, J., et all, Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations, Proceedings of the International Conference on ASPLOS, October, 1994.

[LIPA97] Lipasti, M.H. & Shen, J.P., Superspeculative Microarchitecture for Beyond AD 2000, IEEE Computer Magazine - The Future Processors, V.30, n.9, 1997.

[MIPS95] MIPS R10000 Microprocessor User's Manual, Version 1.0, MIPS Technologies, Inc. North Shoreline, Mountain View, California, June, 1995.

[PARK91] Park, W. W., et al, Performance Advantages of Multithreaded Processors, Proceedings of the International Conference on Parallel Processing, 1991.

[SHAD98] Shade User's Guide, Sun Microsystems Labs, Inc., Mountain View, CA, 1998

[SMIT81] Smith, J. E., A Study of Branch Prediction Strategies, Proceedings of the 8th International Symposium on Computer Architecture - ISCA'81, Minneapolis, Minnesota, May, 1981.

[TRAN92] Tran, T. & Wu, C., Limitation of Superscalar Microprocessor Performance, Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, December, 1992.

[TULL95] Tullsen, D. M., et all, Simultaneous Multithreading: Maximizing On-Chip Parallelism, Proceedings of the ISCA'95, Santa Margherita Ligure, Italy, Computer Architetcure News, n.2, v.23, 1995.

[TULL96] Tullsen, D.M., et all, Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, Proceedings of the 23rd ISCA, Philadelphia, PA, May, 1996.